Chapter 8. Alternative Techniques

Although SOAP is most commonly used for RPC, there are other possibilities as well. SOAP can also be used to create message-style services in which a SOAP envelope is sent to a server without the requirement that anything be returned to the caller. The message can contain any payload that is appropriate for the service — it doesn't have to look like a procedure call. This is similar to the way email systems work. When you send an email, you don't get anything back from the server, and there aren't any rules about what the email can contain. The server accepts your message and does its job without any more involvement from the sender. This is in sharp contrast to RPC services, where a return value is an integral part of the method invocation process.

Not all SOAP implementations support message-style services, but these services can be a useful model and are well worth investigating. GLUE doesn't support message services, but Apache SOAP does, so we'll use Apache SOAP to develop some examples.

The Apache SOAP API for message-style services is a lower level API than the one we've been using for RPC. This shouldn't be surprising, since RPC works by sending messages back and forth. The RPC API is built on top of the message API and includes some useful abstractions that make writing RPC services and service clients relatively easy. You won't find any such convenience in the Apache SOAP message service world. Instead you'll have to work with the lower level APIs that deal directly with envelope marshalling and unmarshalling, XML structures, and a variety of other things that were handled automatically in the RPC world. The upside is that you have more control over the process, although in most cases I prefer better abstractions.

In most of the examples so far, we've used the SOAP encoding style, but that's not the only way to encode data. Literal encoding is another important and useful mechanism, especially when you're trying to pass existing XML documents using SOAP. So let's look at an example using literal encoding to see how this option is used.

We'll also take a look at SOAP with attachments. The idea behind this is simple: just as you can attach a file to an email message, you can attach external data to a SOAP message, using MIME. This technique is useful when you need to pass data that has no business being encoded or decoded as part of your SOAP message, such as an image or sound file. Using attachments lets you send large binary data objects along with your SOAP messages without having to include the data within the SOAP envelope. For many systems, using attachments results in a significant performance improvement over encoding the data within the envelope.

8.1 SOAP Messaging

The format of the SOAP envelope for messaging services differs from the format used for RPC services. There is no concept of a method signature, method parameters, or return values in the messaging model. Basically, you're just sending some XML to a service. However, there are some requirements that allow you to direct the message to the correct service and service method. The name of the first child element of the SOAP `Body` must correspond to the name of the service method to be used, and the `xmlns` namespace attribute on that element specifies the name of the service. Here's a simple SOAP message that is directed to the `recordTemperature` method of the `urn:WeatherDiary` service for processing:

```
<SOAP_ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
      <recordTemperature xmlns="urn:WeatherDiary"/>
      <temperature>75.5</temperature>
      <zipcode>50328</zipcode>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `recordTemperature` service method records the current temperature associated with a zip code. Notice that there's no data in the `recordTemperature` element, although it's perfectly valid for that element to contain data and child elements. The point is that it isn't required. In fact, I could have made the `temperature` and `zipcode` elements children of `recordTemperature` instead of siblings. Either way is fine; as far as SOAP is concerned, this is just a message that is delivered to a method, not an RPC with arguments that need to be handled correctly. As you'll see shortly, we're going to have to write some code to get at these elements either way.

So as long as you've got that first element set up properly, the rest of the XML is up to you. You don't have any parameters to encode or any strict syntax requirements. This doesn't mean you can just send any message to any message service endpoint, however. On the contrary, the absence of strict syntax requirements makes it harder to interoperate with services and service clients not under your control. You have to make sure your messages conform to the formats required by various other services. That doesn't make this service model bad; it makes it more flexible, with all of the problems associated with that level of flexibility.

8.1.1 Writing a Message Service

Writing a message-style service is not all that different from writing an RPC service: you have to write a method that is called when a message is received. The difference is in the method signatures. With RPC, we had the luxury of defining the service method's parameters and their types, as well as the return value type. In the messaging model, all service methods have the same signature. There are three parameters. The first is an instance of `org.apache.soap.Envelope`, which encapsulates the contents of the SOAP envelope that was received by the service method. The other two parameters are instances of `org.apache.soap.rpc.SOAPContext`,[1] a class that encapsulates a MIME multipart message. One is for the input, and the other is for the output. Yes, that's right — the output! Even though SOAP messaging services are not required to provide a response, they are permitted to do so if the underlying transport supports it. HTTP certainly allows content to be returned to the caller, so a message service built on top of HTTP can provide a response. Actually, the Apache SOAP framework requires a message service to provide a response.

Let's write a Java class for the `urn:WeatherDiary` service. We'll need a method called `recordTemperature` so that client applications can send their current temperature to the service. Here is the `javasoap.book.ch8.WeatherDiary` class that implements the service:

[1] It's interesting that this class resides in the SOAP RPC package. That's not an indication that its intended use is only in RPC-style services; it probably should be in a different package.

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import javax.mail.MessagingException;
import org.w3c.dom.*;
public class WeatherDiary {
   Hashtable _diary = new Hashtable(  );
   public void recordTemperature (Envelope env, SOAPContext
                     reqCtx, SOAPContext resCtx)
            throws Exception {

      Vector v = env.getBody().getBodyEntries(  );
      int cnt = v.size(  );
      String zipcode = null;
      String temperature = null;
      for (int i = 0; i < cnt; i++) {
         Element e = (Element)v.elementAt(i);
         String name = e.getTagName(  );
         if (name.equals("zipcode")) {
            zipcode = e.getFirstChild().getNodeValue(  );
         }
         else if (name.equals("temperature")) {
            temperature = e.getFirstChild().getNodeValue(  );
         }
      }
      if (zipcode == null || temperature == null) {
         throw new IllegalArgumentException(
               "ZIPCODE and/or TEMPERATURE Not Specified");
      }
      _diary.put(zipcode, temperature);
      resCtx.setRootPart("OK", "text/xml");
   }
}
```

We want to get at the elements contained within the SOAP body. We do this by calling the `getBody( )` method of the `env` parameter, which returns an instance of `org.apache.soap.Body`. Then we call the body object's `getBodyEntries( )` method, which returns a `java.util.Vector` containing instances of `org.w3c.dom.Element`. Each `Element` in the vector encapsulates one of the direct child elements of the SOAP body. If you look at the SOAP envelope that we're sending to the service, you'll see that we used three direct child elements of the body: `recordTemperature`, `zipcode`, and `temperature`. That's the format we'll use for the `recordTemperature` method of the `urn:WeatherDiary` service. We iterate over the `Vector` to get at each `Element` of the body. To find out what kind of element we got, we call the `getTagName( )` method and check the result to see if it's the `zipcode` or `temperature` element. (We'll also get the `recordTemperature` element used to specify the service and method name, but we don't have any use for that element in this example.) When we find an element tag that we're interested in, we get its first (and only) child node, where we'll find the data for that element. The data is stored in the local variables `zipcode` and `temperature`. After falling out of the loop, we look to see whether either of the variables is still `null`. If so, at least one of the required elements did not appear in the message, so we throw an exception. That exception causes Apache SOAP to return a SOAP fault to the caller. Otherwise, the data is stored in the `_diary` hash table with the zip code as the key and the temperature as the value. We're going to deploy this service using application scope, so the contents of the hash table will be intact across invocations.

The last step is to set a return value. This is a bit confusing — why should a message-style service have a return value? However, you'll get an exception from the Apache framework if you don't supply one. As I said earlier, the Apache framework insists on supplying a response message, and wants something to put in the message. The easiest way to set the return value is to supply a simple text message as the root part of the response context by calling the `resCtx.setRootPart( )` method. The first parameter is the text to be sent; the second part is the MIME type of the data, in this case `text/xml`. All I'm doing is sending back the string "OK" to indicate that the data was received.

Now let's deploy the service. Here's the deployment descriptor:

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:WeatherDiary" type="message">
  <isd:provider
     type="java"
     scope="Application"
     methods="recordTemperature">
    <isd:java
       class="javasoap.book.ch8.WeatherDiary"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
  </isd:mappings>
</isd:service>
```

The only difference between this deployment descriptor and those we created for RPC-style services is the use of the `type` attribute on the `isd:service` element. The value of this attribute is now set to `"message"`, which indicates that the service being deployed is a message service, rather than an RPC service.

So this service can go along happily receiving temperature updates from client applications that supply readings at various locations. Let's take a look at a client application that sends the updates. We'll build the XML for the SOAP envelope and then send it to the `urn:WeatherDiary` service.

```
package javasoap.book.ch8;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;
public class WeatherClient {
   public static void main (String[] args) throws Exception {
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder(  );
      Document doc = xdb.newDocument(  );
```

```
        Element elem =
               doc.createElementNS(Constants.NS_URI_SOAP_ENV,
               "SOAP-ENV:Envelope");
        doc.appendChild(elem);

        Element sub =
               doc.createElementNS(Constants.NS_URI_SOAP_ENV,
               "SOAP-ENV:Body");
        elem.appendChild(sub);
        elem = sub;
        sub = doc.createElement("recordTemperature");
        sub.setAttribute("xmlns", "urn:WeatherDiary");
        elem.appendChild(sub);
        sub = doc.createElement("zipcode");
        Text txt = doc.createTextNode("");
        txt.setData("12345");
        sub.appendChild(txt);
        elem.appendChild(sub);
        sub = doc.createElement("temperature");
        txt = doc.createTextNode("");
        txt.setData("52.3");
        sub.appendChild(txt);
        elem.appendChild(sub);
        Envelope msgEnv = Envelope.unmarshall(
                             doc.getDocumentElement(  ));
        URL url = new URL(
          "http://georgetown:8080/soap/servlet/messagerouter");
        Message msg = new Message(  );
        msg.send(url, "", msgEnv);

        SOAPTransport st = msg.getSOAPTransport(  );
        BufferedReader br = st.receive(  );
        String line;
        while ((line = br.readLine(  )) != null) {
           System.out.println(line);
        }
    }
}
```

Most of this code is not really about SOAP; it's about building an XML document that happens to be a SOAP envelope. However, it seems to me that if you're going to use the Apache message-style services, then you may also be creating your SOAP envelope on the fly.[2] So let's take a quick look at how it's done.

The first step is to get an instance of `org.w3c.dom.Document`, which encapsulates the entire XML document starting with the root element. We get the initial `Document` by calling `XMLParserUtils.getXMLDocBuilder( )` to get a `DocumentBuilder`, and then calling its `newDocument( )` method. Now we can work on the envelope element, which we will use as the root of our document. The `doc.createElementNS( )` method creates an element along with a qualifying namespace identifier. The first parameter is the standard SOAP envelope namespace `http://schemas.xmlsoap.org/soap/envelope/`, which we get from `Constants.NS_URI_SOAP_ENV` . The second parameter is a qualified name, i.e., a qualifier and element name separated by a colon (:). In this case we're using the qualifier `SOAP-ENV` to be consistent with earlier examples, but the qualifier is nothing more than a namespace

[2] If you won't be creating your own envelope on the fly, you might want to look at the messaging example provided with Apache SOAP. That example assumes that the envelope was generated by some other system and shows you how to load it into your application to be sent to a service.

identifier; you can use any name. The name of the element itself is `Envelope`. Now that we've created the SOAP envelope, we need to add it to the document by calling `doc.appendChild( )`. We use the same technique to create the `Body` element, only this time we append it to the `Envelope` element rather than to the document itself, because the SOAP `Body` is a child element of the SOAP `Envelope`.

Now that we've set up the `Envelope` and the `Body`, we need to create elements for our specific content and add them to the body. So we set the `elem` variable to the `Body` element, and use the `sub` variable to create the child elements of the body. To invoke the correct service method, the first child element of the body must be named `recordTemperature`. This element is created by calling `doc.createElement( )` and passing the name of the element as the parameter. The resulting element is saved in the `sub` variable. To invoke the proper service, we need to set the `xmlns` attribute value to the name of the service. We do this by calling `sub.setAttribute( )`, where the first parameter is the name of the attribute and the second parameter is the value. The `recordTemperature` element doesn't have any data or child elements, so we just append it to the body by calling `elem.appendChild(sub)`.

Next we create the `zipcode` element, again using the `doc.createElement( )` method. No attributes are necessary for this element, but we need to add a data node for the zip code. We create an instance of `org.w3c.dom.Text` by calling the `doc.createTextNode( )` method. The `Text` class is used to represent textual data, which is perfect for our purpose. The zip code data is set by calling `txt.setData( )`, and the text node is appended to the `zipcode` element by the `sub.appendChild( )` method. In turn, the `zipcode` element is appended to the `Body` element. We create the `temperature` element in exactly the same way, and then it too is appended to the `Body` element.

We're going to send this message to the server using an instance of `org.apache.soap.messaging.Message`, but first we need to turn the XML document into an instance of `org.apache.soap.Envelope`. We accomplish this by calling the `Envelope.unmarshall( )` method, passing it the root document element. This seems like it's going to add some overhead; you can bet that the resulting `Envelope` object will be marshalled all over again to be sent to the server. In any case, the result of this method call is an `Envelope` instance called `msgEnv`. Next we create an instance of `java.net.URL` with the address of the SOAP router that handles routing for message services. For RPC services, we've been sending SOAP requests to *http://georgetown:8080/soap/servlet/rpcrouter*; for message services, we send to *http://georgetown:8080/soap/servlet/messagerouter*. Now we can create an instance of `Message` and call its `send( )` method. The URL is the first parameter, and the envelope is the last. The second parameter is an empty string representing the SOAP actor; since we don't have any use for a SOAP actor here, an empty string is good enough.

Because we know that our service transport is HTTP and that our service responds with either a fault or a simple "OK", it's a good idea to look at the response before the application terminates. To view the response, we start by calling `msg.getSOAPTransport( )`, which gives us an instance of the transport used to send the message. This method returns an instance of `org.apache.soap.transport.SOAPTransport`. If the transport supports responses, like HTTP does, you can call the transport's `receive( )` method, which returns a `BufferedReader` object. If the transport doesn't support responses, the `receive( )` method returns `null`. Once we have the `BufferedReader`, all we do is grab each line of the response

and print it to the console. If the service was happy with the message, the only thing that will be printed is the string "OK". Otherwise, you'll see the entire SOAP fault document.

Now that we've put the client together, let's look at the SOAP envelope that it sends:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <recordTemperature xmlns="urn:WeatherDiary"/>
    <zipcode>12345</zipcode>
    <temperature>52.3</temperature>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Now we have a service that can collect information and a client that can send information, but we still don't have a mechanism that allows someone to retrieve the data that has been collected. We should really extend the service by adding a method for querying the temperature at a specified zip code. But wait . . . that sounds more like an RPC-style service than a messaging service. It's one thing to have a message service respond with something general like "OK, I got your message," but it's quite another to have it respond with a specific result based on the content of the message received. Why should you use RPC-style services when working with a request/response model? Many developers hold strong opinions about what kind of service is appropriate for each application, but other than the fact that not all messaging transports support a response, there aren't any good reasons not to use a message service in this situation. It's easy to fall into the trap of thinking only in a procedural model. Some systems could benefit from the freedom provided by the messaging model, where you're not bound by the rules of specific parameter passing like you are in the RPC world. For example, you might want to send a purchase order document to a message service and receive a confirmation document with a delivery schedule in return. This exchange could be implemented as an RPC, but it's a lot easier to think of it in terms of a message exchange. Although the structure of SOAP RPC doesn't prevent you from developing open-ended parameter passing models, the messaging model may fit your designs better. So let's expand our service with a `getTemperature( )` method that returns an XML document.

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import javax.mail.MessagingException;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
public class WeatherDiary {
   Hashtable _diary = new Hashtable(  );
   Hashtable _times = new Hashtable(  );
   public void getTemperature(Envelope env, SOAPContext reqCtx,
                              SOAPContext resCtx)
         throws Exception {
      Vector v = env.getBody().getBodyEntries(  );
      int cnt = v.size(  );
      String zipcode = null;
      for (int i = 0; i < cnt; i++) {
         Element e = (Element)v.elementAt(i);
         String name = e.getTagName(  );
```

```
         if (name.equals("zipcode")) {
            zipcode = e.getFirstChild().getNodeValue(  );
         }
      }
      if (zipcode == null) {
         throw new IllegalArgumentException(
                    "ZIPCODE Not Specified");
      }
      String temperature = (String)_diary.get(zipcode);
      String rectime = (String)_times.get(zipcode);
      String response = "<WeatherDiaryResponse>" +
                        (char)10 +
                        "   <temperature>" + temperature +
                        "</temperature>" +
                        (char)10 +
                        "   <recordTime>" + rectime +
                        "</recordTime>" +
                        (char)10 +
                        "</WeatherDiaryResponse>";
      resCtx.setRootPart(response, "text/xml");
   }
   public void recordTemperature (Envelope env, SOAPContext
                    reqCtx, SOAPContext resCtx)
        throws Exception {

      Vector v = env.getBody().getBodyEntries(  );
      int cnt = v.size(  );
      String zipcode = null;
      String temperature = null;
      for (int i = 0; i < cnt; i++) {
         Element e = (Element)v.elementAt(i);
         String name = e.getTagName(  );
         if (name.equals("zipcode")) {
            zipcode = e.getFirstChild().getNodeValue(  );
         }
         else if (name.equals("temperature")) {
            temperature = e.getFirstChild().getNodeValue(  );
         }
      }
      if (zipcode == null || temperature == null) {
         throw new IllegalArgumentException(
               "ZIPCODE and/or TEMPERATURE Not Specified");
      }
      _diary.put(zipcode, temperature);
      _times.put(zipcode, new Date().toString(  ));
      resCtx.setRootPart("OK", "text/xml");
   }
}
```

To make the response from the service method more interesting, we modified the `recordTemperature( )` method to store the time the temperature was recorded in another hash table called `_times`, which uses the zip code as the key and the current time as the value. But the most important addition is the `getTemperature( )` service method. This method starts by looking through the child elements of the SOAP body for an element called `zipcode`, which tells the method which zip code is being requested. If a `zipcode` element isn't found, the method throws an exception, resulting in a SOAP fault. Once we have a zip code, we retrieve the temperature and the time it was recorded from the `_diary` and `_times` hash tables, respectively. Next, we build an XML document in the `String` variable called `response`. This document is simple. Its root is an element named `WeatherDiaryResponse`,

which contains two child elements named `temperature` and `recordTime`. These elements contain the temperature and the time it was recorded for the specified zip code. The `response` string is used as the first parameter of the `resCtx.setRootPart( )` method. The result is that we've responded to the incoming message with an XML document that contains the requested data. You'll need to redeploy the service, adding the `getTemperature( )` method to the deployment descriptor.

Now let's create a client application that requests temperature data. The client goes through the same process that we've demonstrated in other examples. This time, we include only the `zipcode` element and its data, and the first element is named `getTemperature` to invoke the correct service method. Here's the code for the client:

```
package javasoap.book.ch8;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;
public class TemperatureClient {
   public static void main (String[] args) throws Exception {
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder(  );
      Document doc = xdb.newDocument(  );

      Element elem =
                doc.createElementNS(Constants.NS_URI_SOAP_ENV,
                "SOAP-ENV:Envelope");
      doc.appendChild(elem);

      Element sub =
                doc.createElementNS(Constants.NS_URI_SOAP_ENV,
                "SOAP-ENV:Body");
      elem.appendChild(sub);
      elem = sub;
      sub = doc.createElement("getTemperature");
      sub.setAttribute("xmlns", "urn:WeatherDiary");
      elem.appendChild(sub);
      sub = doc.createElement("zipcode");
      Text txt = doc.createTextNode("");
      txt.setData("12345");
      sub.appendChild(txt);
      elem.appendChild(sub);
      Envelope msgEnv = Envelope.unmarshall
                           (doc.getDocumentElement (  ));
      URL url = new URL(
          "http://georgetown:8080/soap/servlet/messagerouter");
      Message msg = new Message (  );
      msg.send (url, "", msgEnv);

      SOAPTransport st = msg.getSOAPTransport (  );
      BufferedReader br = st.receive (  );
      String line;
```

```
      while ((line = br.readLine (  )) != null) {
         System.out.println (line);
      }
   }
}
```

Before you run this application, you'll need to populate the service with temperature data for zip code 12345. (Redeploying the service with the new `getTemperature( )` method destroys the data we saved earlier.) After putting some data into the service, run the client. It should spit out the following:

```
<WeatherDiaryResponse>
   <temperature>52.3</temperature>
   <recordTime>Wed Oct 31 14:58:48 EST 2001</recordTime>
</WeatherDiaryResponse>
```

We won't look at the SOAP envelope that was sent to the `getTemperature( )` method, since you can figure out what it looks like by now. The only interesting part of the envelope is the response.

It would be nice to provide some more structure to the output; right now, we just dump a few XML elements. So let's build a document out of the response instead. The following code replaces everything after the `msg.send( )` call. After we get the `BufferedReader`, we still need an instance of `DocumentBuilder`. But now, instead of building a document piece by piece, we'll use the content of the response instead. We create an instance of `org.xml.sax.InputSource`, passing the `BufferedReader` to the constructor. Now we call `xdb.parse( )` with the `InputSource` as the parameter. This parses the XML in the response and builds a `Document` instance for us. After getting the root element, we print it so that we can be sure we're getting the element we're expecting. Then we get a list of all of the child elements of the root by calling `getElementsByTagName("*")` , passing it the wildcard (`*`). This method retrieves all of the child elements of the `WeatherDiaryResponse` root element. Finally, the list is traversed and each element is printed, with its associated data.

```
SOAPTransport st = msg.getSOAPTransport (  );
BufferedReader br = st.receive (  );

xdb = XMLParserUtils.getXMLDocBuilder(  );
InputSource is = new InputSource(br);
doc = xdb.parse(is);
elem = doc.getDocumentElement(  );
System.out.println("Root Element Is: " +
                              elem.getTagName(  ));

NodeList children = elem.getElementsByTagName("*");
int cnt = children.getLength(  );
for (int i = 0; i < cnt; i++) {
   Node n = children.item(i);
   String name = n.getNodeName(  );
   String val = n.getFirstChild().getNodeValue(  );
   System.out.println(name + ": " + val);
}
```

Running the modified client produces the following output:

```
Root Element Is: WeatherDiaryResponse
temperature: 52.3
recordTime: Wed Oct 31 14:58:48 EST 2001
```

So with a little bit of XML manipulation we can return meaningful data from a message-style service, assuming the underlying transport supports the ability to provide a response.

8.2 Literal Encoding

We've seen how to use SOAP for RPC-type services, as well as how to pass around XML data in the less restrictive world of SOAP messaging services. But another possibility is to pass XML using RPC services. The problem is that RPC services rely on a type mapping mechanism to convert between XML elements and Java classes. It would be nice if you could use an RPC-style method invocation to pass parameters, with the result being standard XML. Well, you can. To do it, you need to use XML literal encoding, which tells the underlying SOAP framework that the data is XML and should remain that way.

Let's go back to the world of finance. We'll create a service that provides corporate information on companies in an XML format. The service is called `urn:CorporateDataService`, which has a method called `getDataForSymbol` . The service is an RPC service, and the method takes a stock symbol as the parameter. The data returned can include the company name, company address, year of incorporation, number of employees, and the high and low trading price for the past 52 weeks. This service will pull its data from another system that already keeps corporate data in XML format in a database. So it's not the responsibility of the service to format the data, only to pass it back to the caller. This scenario is perfect for using XML literal encoding. If the service were going to process the data in some way, it might just as well be designed to return a complex type that can be mapped to a Java class. Another reason for using literal encoding is that the client application may want to get the data in the XML format used on the back-end. It would be a waste of time to convert the XML on the server side, only to have the client put it back into XML. In this case, we're using SOAP to provide access to existing data, and then relying on existing software systems to do the processing.

To create a Java method with an appropriate signature, we need a class that encapsulates an XML document, since that's what we want to return to the caller. `org.w3c.dom.Element` does what we want, so we'll define our service method to return an instance of that class. The `javasoap.book.ch8.CorporateDataService` class in the following listing implements the service. The service opens a file whose name is the symbol passed to it with the extension *.xml*. The file is expected to be in the same directory as the class itself. (We're assuming that some other system has already generated the corporate data XML files that the service loads.) A `java.io.FileReader` is created, using the filename as the parameter to its constructor. Next we create an instance of `DocumentBuilder` , and call its `parse( )` method to turn the contents of the file into an XML document object. The parameter passed to the `parse( )` method is an instance of `org.xml.sax.InputSource`, which we create using the `FileReader` as the constructor's parameter. Now all we need to do is return the root element, which we can retrieve by calling `doc.getDocumentElement( )` . You can deploy this service just like the other services in this book, listing `getDataForSymbol` as the sole service method. No special mappings are needed.

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
public class CorporateDataService {
   public CorporateDataService(  ) {
   }
   public Element getDataForSymbol(String symbol)
          throws Exception {
      FileReader fr = new FileReader(symbol + ".xml");
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder(  );
      Document doc = xdb.parse(new InputSource (fr));
      if (doc == null) {
         throw new SOAPException(Constants.FAULT_CODE_SERVER,
                           "Invalid Data");
      }
      return doc.getDocumentElement(  );
   }
}
```

Let's take a look at a corporate data file that contains a sample of the XML that our service will return to the caller:

```
<corporateData>
   <symbol>MINDSTRM</symbol>
   <name>MindStream Software, Inc.</name>
   <address>
      <address1>111 Smithtown Bypass</address1>
      <address2>Suite 208</address2>
      <city>Hauppauge</city>
      <state>NY</state>
      <zip>11788</zip>
   </address>
   <incorpYear>1998</incorpYear>
   <numEmployees>unknown</numEmployees>
   <tradeHistory>
      <yearlyHigh>75.5</yearlyHigh>
      <yearlyLow>60</yearlyLow>
   </tradeHistory>
</corporateData>
```

Now we can write a client application to invoke the `getDataForSymbol` method:

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
```

```
public class GetDataClient {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");
      Call call = new Call(  );
      call.setTargetObjectURI("urn:CorporateDataService");
      call.setMethodName("getDataForSymbol");
      call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
      String symbol = "MINDSTRM";
      Vector params = new Vector(  );
      params.addElement(new Parameter("symbol", String.class,
                  symbol, Constants.NS_URI_SOAP_ENC));
      call.setParams(params);
      Response resp;
      try {
         resp = call.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage(  ));
         return;
      }
      if (!resp.generatedFault(  )) {
         Parameter ret = resp.getReturnValue(  );
         Element bookEl = (Element)ret.getValue(  );
         System.out.println(DOM2Writer.nodeToString(bookEl));
      }
   }
}
```

We need to set the encoding style to XML literal encoding, both because we want the server to serialize its response as a literal XML document, and because we want the response to be interpreted that way. To do this, we call `call.setEncodingStyleURI( )` with the argument `Constants.NS_URI_LITERAL_XML` . In Apache SOAP, this sets the encoding style for the `Body` element. That's the highest level at which the `encodingStyle` attribute will be set. However, we don't want the parameter that we're passing (the stock symbol) to be interpreted as XML; after all, it's just a string. We want to use the standard SOAP encoding for the parameter, so we want the `encodingStyle` attribute to appear on the `symbol` element as well, with the appropriate value for SOAP encoding. To specify the encoding style for this parameter, we set the fourth argument of the `Parameter` constructor to `Constants.NS_URI_SOAP_ENC`. In all of the previous examples, we've set this value to `null`, which means that the `encodingStyle` attribute need not appear at the parameter level; the encoding style for the parameter defaults to the encoding style set for the body. Setting the encoding style for the `symbol` element explicitly overrides the body's encoding style.

Now we invoke the method as usual, but we cast the return value to an instance of `Element`, since we're expecting the return value to be literal XML. I pass the resulting `Element` to the `DOM2Writer.nodeToString( )` method so that I can print the XML to the console. Let's take a look at the SOAP envelope that was sent to the service:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getDataForSymbol
xmlns:ns1="urn:CorporateDataService"
SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<symbol xsi:type="xsd:string"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
MINDSTRM
</symbol>
</ns1:getDataForSymbol>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As you can see, the `encodingStyle` attribute is set to literal XML encoding for the SOAP `Body` element. The `symbol` element overrides this setting to use SOAP encoding. Now, here's the response envelope. You can see that the return value uses XML literal encoding, as the entire response is XML data.

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getDataForSymbolResponse
xmlns:ns1="urn:CorporateDataService"
SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<return>
<corporateData>
   <symbol>MINDSTRM</symbol>
   <name>MindStream Software, Inc.</name>
   <address>
      <address1>111 Smithtown Bypass</address1>
      <address2>Suite 208</address2>
      <city>Hauppauge</city>
      <state>NY</state>
      <zip>11788</zip>
   </address>
   <incorpYear>1998</incorpYear>
   <numEmployees>unknown</numEmployees>
   <tradeHistory>
      <yearlyHigh>75.5</yearlyHigh>
      <yearlyLow>60</yearlyLow>
   </tradeHistory>
</corporateData>
</return>
</ns1:getDataForSymbolResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You can use the same technique if you want to send literal XML as a parameter to an RPC service method. Just specify the `Constants.NS_URI_LITERAL_XML` encoding style when you call the `Parameter` constructor. There's no requirement that you use the same encoding style for all the parameters; do whatever is appropriate to your application.

8.2.1 SOAP with Attachments

Many SOAP implementations allow you to send data along with the SOAP envelope as an attachment. Attachments are useful when the data would put a large and unnecessary burden on the SOAP processor. For example, consider how you would send an audio or video file as part of a SOAP message. These can be very large files, and they are not encoded in XML. You could certainly include them in your SOAP envelope using a binary encoding scheme like base64, but that would cause the data to be loaded and processed by your SOAP engine. It would certainly work, but it's a lot of overhead for no gain: why should your SOAP engine process a huge amount of binary data that it can't do anything with? Attaching the data allows you to send it along without incurring the overhead, but also without the structure of XML.

Let's create a buying and selling service for used cars. We'll have methods that list a car for sale and that retrieve information about available cars. We need a custom type to represent a listing; this type encapsulates the details of the car and the contact information for the car's seller. Here's the code for the `javasoap.book.ch8.Listing` class that we'll use to pass this information. It includes a `toString( )` method so we can easily display its contents.

```
package javasoap.book.ch8;
public class Listing {
   String _make;
   String _model;
   int    _year;
   int    _miles;
   String _ownerName;
   String _ownerEmail;
   public Listing(  ) {
   }
   public Listing(String make, String model,
     int year, int miles, String ownerName, String ownerEmail) {
      setMake(make);
      setModel(model);
      setYear(year);
      setMiles(miles);
      setOwnerName(ownerName);
      setOwnerEmail(ownerEmail);
   }
   public void setMake(String make) {
      _make = make;
   }
   public String getMake(  ) {
      return _make;
   }
   public void setModel(String model) {
      _model = model;
   }
   public String getModel(  ) {
      return _model;
   }
   public void setYear(int year) {
      _year = year;
   }
   public int getyear(  ) {
      return _year;
   }
```

```
    public void setMiles(int miles) {
        _miles = miles;
    }
    public int getMiles(  ) {
        return _miles;
    }
    public void setOwnerName(String ownerName) {
        _ownerName = ownerName;
    }
    public String getOwnerName(  ) {
        return _ownerName;
    }
    public void setOwnerEmail(String ownerEmail) {
        _ownerEmail = ownerEmail;
    }
    public String getOwnerEmail(  ) {
        return _ownerEmail;
    }
    public String toString(  ) {
        String result = String.valueOf(_year) + " " +
                        _make + " " + _model +
                        " with " + String.valueOf(_miles) +
                        " miles:" +
                        "\n   Owned by " + _ownerName + "(" +
                        _ownerEmail + ")";
        return result;
    }
}
```

The `Listing` class has string properties called `Make` and `Model`, which represent the manufacturer of the car and the model name, respectively. The `Year` is an integer property representing the year the car was built, and the `Miles` integer property is the mileage of the car. The `OwnerName` and `OwnerEmail` are string properties for the name and email address of the person who owns the car. The get and set methods for the properties conform to the JavaBeans patterns, so we won't need to write a custom serializer for the `Listing` class.

Now let's create a class to implement the `urn:UsedCarListingService` service. sending listings to the service. The `addListing( )` method,[3] which sends listings to the service, takes an instance of `Listing` as its first parameter and returns an integer that represents the listing's ID. We'll allow the caller to attach an image file to an invocation of the `addListing( )` web service method, so the second parameter is an instance of `javax.activation.DataHandler` that encapsulates the attachment. Apache SOAP already has a serializer for the `DataHandler` class, and takes advantage of the JavaBeans Activation Framework (JAF) for handling MIME attachments. This framework makes receiving attachments in an RPC service as simple as passing a parameter. The work of referencing and extracting the attachment is done for you, and you can work with the data through the JAF API. The `search( )` method has `String` parameters for the make and model of the car, and returns an array of listing IDs. And finally, a `getListing( )` method takes a listing ID as its parameter and returns an instance of `Listing`, and a `getImage( )` method also takes the listing ID as its parameter and returns the associated image file as an attachment. Here is the `javasoap.book.ch8.UsedCarListingService` class:

[3] Of course a `removeListing` would be a good idea too, but it's not important for the example.

```
package javasoap.book.ch8;
import java.util.*;
import java.io.*;
import javax.activation.*;
import org.apache.soap.util.mime.*;
public class UsedCarListingService {
   Hashtable _listings = new Hashtable(  );
   int _nextId = 100;
   public int addListing(Listing listing, DataHandler handler)
               throws Exception {
      _listings.put(new Integer(_nextId), listing);
      int listno = _nextId;
      _nextId++;
      if (handler != null) {
         String fname = String.valueOf(listno) + ".bin";
         DataSource ds = handler.getDataSource(  );
         ByteArrayDataSource bsource =
               new ByteArrayDataSource(ds.getInputStream(  ),
                                       handler.getContentType(  ));
         bsource.writeTo(new FileOutputStream(fname));
      }
      return listno;
   }
   public Integer[] search(String make, String model) {
      Vector v = findListingIds(make, model);
      int cnt = v.size(  );
      if (cnt == 0)
         return null;
      Integer ids[] = new Integer[cnt];
      for (int i = 0; i < cnt; i++) {
         ids[i] = (Integer)v.elementAt(i);
      }
      return ids;
   }
   protected Vector findListingIds(String make, String model) {

      Vector result = new Vector(  );
      for (Enumeration e = _listings.keys(  ) ;
                           e.hasMoreElements(  ) ;) {
         Integer i = (Integer)e.nextElement(  );
         Listing listing = (Listing)_listings.get(i);
         if (make.equals(listing.getMake(  )) &&
             model.equals(listing.getModel(  ))) {
            result.add(i);
         }
      }
      return result;
   }
   public Listing getListing(int id) {
      Listing listing = (Listing)_listings.get(new Integer(id));
      return listing;
   }
   public DataHandler getImage(int id)
        throws Exception {
      String fname = String.valueOf(id) + ".bin";
      DataSource ds =
           new ByteArrayDataSource(new File(fname), null);
      DataHandler dh = new DataHandler(ds);
      return dh;
   }
}
```

The service operates at application scope, so all of the listings are available as long as the service is running. We don't bother to keep the listings in any kind of persistent storage, although you obviously would in the real world. The attached files, however, are stored in files on disk. We'll keep all of the listings in a hash table called `_listings`, where the keys are the unique listing IDs. New ID values are generated by incrementing the value of `_nextId` each time the `addListing( )` method is invoked. The first part of `addListing( )` stores the listing passed as a parameter in the hash table. The key to the hash table is an `Integer` based on the value of `_nextId`, which is subsequently incremented. The next step is to extract the attachment and store its contents in a local file. The filename is just the listing number with a *.bin* extension. We get a reference to a `javax.activation.DataSource` by calling the `getDataSource( )` method of the `handler` parameter, and then create an `org.apache.soap.util.mime.ByteArrayDataSource`. The first parameter to its constructor is a `java.io.InputStream` for the data, which we get by calling `ds.getInputStream( )`. The other parameter is a `String` containing the MIME type of the data, which we get by calling `handler.getContentType( )`. Now we can save the contents of the attachment in a file by calling the `writeTo( )` method of the `ByteArrayDataSource`, passing it an instance of `java.io.FileOutputStream` based on the filename we generated earlier. Now that the attachment data has been stored locally, we just return the listing number to the caller.

The `search( )` method calls a local method called `findListingIds( )`, which returns a `Vector` containing `Integer` objects. Note that `findListingIds( )` is not exposed as a service method. Each entry in the `Vector` represents the ID of a listing that matches the specified make and model. Then an `Integer[]` is allocated, populated with the values in the `Vector`, and returned to the caller. This array allows the caller to use the list of IDs to retrieve the actual listings as well as the associated images.

The `getListing( )` method takes a single integer parameter, the listing ID. This ID is used to get the `Listing` object from the `_listings` hash table. The `Listing` is then returned to the caller. The `getImage( )` method also takes the listing ID as an integer parameter, and uses it to generate the filename for the image associated with the listing. That name is used to instantiate a `java.io.File`, which is passed to the constructor of a `ByteArrayDataSource`. It's not necessary to specify the content type of the data, so `null` is used as the second parameter. A new `DataHandler` is created using the data source; this handler is returned to the caller. Now you can deploy the service, exposing the `addListing`, `search`, `getListing`, and `getImage` service methods. Remember to add a mapping entry for the custom listing type, as shown:

```
  <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="urn:UsedCarListingService" qname="x:Listing"
             javaType="javasoap.book.ch8.Listing"
java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

Now let's work on a client application that sends a listing to the service. We won't send a large binary file because we want to verify that the data is correct, so we'll use a small human-readable file named *data.bin*:

```
1:Rob
2:Englander
3:MindStream Software, Inc.
```

Here's the code for the client application:

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.apache.soap.util.mime.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.util.xml.*;
public class AddListingClient {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
           "http://georgetown:8080/soap/servlet/rpcrouter");
      Call call = new Call(  );
      call.setTargetObjectURI("urn:UsedCarListingService");
      call.setMethodName("addListing");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      SOAPMappingRegistry smr = new SOAPMappingRegistry(  );
      BeanSerializer beanSer = new BeanSerializer(  );
      call.setSOAPMappingRegistry(smr);
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
         new QName("urn:UsedCarListingService", "Listing"),
                javasoap.book.ch8.Listing.class,
                beanSer, beanSer);
      Vector params = new Vector(  );
      Listing listing = new Listing("Pontiac", "Firebird", 1968,
             105000, "Rob Englander", "rob@mindstrm.com");
      params.addElement(new Parameter("listing",
                javasoap.book.ch8.Listing.class, listing, null));
      DataSource ds = new
             ByteArrayDataSource(new File("data.bin"), null);
      DataHandler dh = new DataHandler(ds);
      params.addElement(new Parameter("handler",
                javax.activation.DataHandler.class, dh, null));
      call.setParams(params);

      Response resp;
      try {
         resp = call.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage(  ));
         return;
      }
      if (!resp.generatedFault(  )) {
         Parameter ret = resp.getReturnValue(  );
         System.out.println("The listing number is " +
                               ret.getValue(  ));
      }
      else {
         Fault fault = resp.getFault(  );
         System.out.println ("Fault Code = " +
                fault.getFaultCode(  ));
```

```
            System.out.println ("      String = " +
                    fault.getFaultString(  ));
        }
    }
}
```

The client application starts by setting up the call to invoke the `addListing` service method. Since we're going to pass an instance of `Listing`, we map it to an instance of `BeanSerializer`. A listing is created for a 1968 Pontiac Firebird with 105,000 miles on it, owned by me. This is the first parameter for the method invocation. Next, an instance of `ByteArrayDataSource` is created. The first parameter is a `java.io.File` based on the *data.bin* filename. Once again, we don't specify the content type, so `null` is used for the second parameter. The data source is then used to construct a `DataHandler` object, which serves as the second parameter of the method invocation.

Run this example and you'll get the following output:

```
The listing number is 100
```

Since the service is application scoped, the listing number will increase by 1 each time you run the client.

The SOAP envelope sent to the server does not contain the contents of the image file. The transport is using a MIME multipart message in which the SOAP envelope is the first part and the attached file contents are the second part. So the XML element for the `handler` parameter is a reference to the second part, and looks like this:

```
<handler href ="cid:6889270.1004989967097.apache-soap.georgetown"/>
```

When you run the example, the value of the reference will be different because it's generated on the fly and includes the name of the source (here, the source is `georgetown`). But the important part is that the contents of the attached file do not appear within the SOAP envelope. In this example those contents are small, but imagine if we attached a 10 MB audio file, or a video clip of the car running in a stock car race. If you were to snoop at the HTTP stream sent to the server for this invocation, you'd see the following after the close of the SOAP envelope.

```
Content-Type: application/octet-stream
Content-Transfer-Encoding: 8bit
Content-ID: <6889270.1004989967097.apache-soap.georgetown>
Content-Length: 52

1: Rob
2: Englander
3: MindStream Software, Inc.
```

This is the encoded content of the attachment. In a real application, the attachment would probably be a large stream of encoded binary data, but in this program it's just a short message that allows us to verify that the attachment was handled correctly.

Now let's write a client that retrieves the listings and their associated image files. First we'll invoke the `search` method, passing "Pontiac" and "Firebird" as the parameters. Assuming

we've run `AddListingClient` only once, we should get back an integer array with a single element of value 100. This is the ID for the first and only listing. We'll use the ID to retrieve the listing as well as the image file (as an attachment). Here is the code:

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.apache.soap.util.mime.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.util.xml.*;
public class SearchClient {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");
      Call call = new Call(  );
      call.setTargetObjectURI("urn:UsedCarListingService");
      call.setMethodName("search");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      SOAPMappingRegistry smr = new SOAPMappingRegistry(  );
      BeanSerializer beanSer = new BeanSerializer(  );
      call.setSOAPMappingRegistry(smr);
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
          new QName("urn:UsedCarListingService", "Listing"),
          javasoap.book.ch8.Listing.class, beanSer, beanSer);
      Vector params = new Vector(  );
      String make = "Pontiac";
      String model = "Firebird";
      params.addElement(new Parameter("make",
                java.lang.String.class, make, null));
      params.addElement(new Parameter("model",
                java.lang.String.class, model, null));
      call.setParams(params);

      Response resp;
      try {
         resp = call.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage(  ));
         return;
      }
      if (!resp.generatedFault(  )) {
         Parameter ret = resp.getReturnValue(  );
         int[] ids = (int[])ret.getValue(  );
         int cnt = ids.length;
         for (int i = 0; i < cnt; i++) {
            call.setMethodName("getListing");
            params = new Vector(  );
            params.addElement(new Parameter("id",
                java.lang.Integer.class, new Integer(ids[i]),
                null));
            call.setParams(params);
```

```
            try {
               resp = call.invoke(url, "");
            }
            catch (SOAPException e) {
               System.out.println(e.getMessage( ));
               return;
            }
            ret = resp.getReturnValue( );
            Listing listing = (Listing)ret.getValue( );
            System.out.println(listing);
            call.setMethodName("getImage");
            try {
               resp = call.invoke(url, "");
            }
            catch (SOAPException e) {
               System.out.println(e.getMessage( ));
               return;
            }
            ret = resp.getReturnValue( );
            DataHandler handler = (DataHandler)ret.getValue( );
            DataSource ds = handler.getDataSource( );
            String fname = "EX17." +
                     String.valueOf(ids[i]) + ".bin";
            ByteArrayDataSource bsource =
                  new ByteArrayDataSource(ds.getInputStream( ),
                                          handler.getContentType( ));
            bsource.writeTo(new FileOutputStream(fname));
            System.out.println(
                  "   Image File Stored In " + fname);
         }
      }
      else {
         Fault fault = resp.getFault( );
         System.out.println ("Fault Code = " +
                 fault.getFaultCode( ));
         System.out.println ("      String = " +
                 fault.getFaultString( ));
      }
   }
}
```

The first part of the code should be really familiar by now. We set up the call for the `search` method, map the `Listing` type, and add two string parameters for searching for listings of Pontiac Firebirds. The return value of the call is cast to an `int[]`. (Even though the service class returns an `Integer[]`, the Apache SOAP framework maps an array of primitive data elements to a corresponding array of primitive Java types, so we get back an `int[]`.) Now we can loop over the array of listing IDs. For each one, we set the method name to `getListing`, and we use the listing ID from the `ids` array to create a `Parameter` for the call. Now we invoke the method, this time casting the result to an instance of `Listing`. Since `Listing` implements a `toString( )` method, we can just pass the `listing` object to `System.out.println( )` to display its contents. Next, we set the method name to `getImage( )` and invoke again, this time casting the return value to a `DataHandler` and getting its `DataSource`. We write the contents of the attachment to a local file using the same technique we used in the `UsedCarListingService` class. The filename takes the form *EX17.xxx.bin*, where *xxx* is the listing ID.

The last step is to display the name of the file that contains the contents for the given listing. Here's what the output looks like:

```
1968 Pontiac Firebird with 105000 miles:
   Owned by Rob Englander(rob@mindstrm.com)
   Image File Stored In EX17.100.bin
```

If you look at the contents of the file *EX17.100.bin*, you'll find that it's identical to the contents of the original attachment:

```
1: Rob
2: Englander
3: MindStream Software, Inc.
```

If you want to work with attachments in Apache SOAP message-style services, you have to work at a lower level, without the nice abstractions provided at the RPC level of the API. GLUE also supports SOAP attachments using a straightforward API. Although we won't cover that here, I'm sure you'll find the GLUE abstractions easy to understand and use.

The alternative SOAP mechanisms we've discussed in this chapter, used along with all of the other techniques we've covered, allow you to use SOAP for just about any kind of data transfer model.